THE ISLAMIC UNIVERSITY – FACULTY OF ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

LCOM 3010 – Introduction to Computer Architecture Lab

Lab #5
INSTRUCTIONS AND DATAPATH

BY

Eng: Rasha Ghazy Sammour

Submitted for

Dr. Mohammed Mikki

March 31, 2021

Single Cycle Data Path

Process:

In lab 1, we reminded ourselves that the *datapath* and *control* are the two components that come together to be collectively known as the processor.

- Datapath consists of the functional units of the processor.
  - Elements that hold data (Program counter, register file, instruction memory, etc.)
  - Elements that operate on data (ALU, adders, etc.)
  - Buses for transferring data between elements.
- Control commands the datapath regarding when and how to route and operate on data.

MIPS:

To showcase the process of creating a datapath and designing a control, we will be using a subset of the MIPS instruction set. Our available instructions include:

- `add, sub, and, or, slt`
- `lw, sw`
- `beq, j`

DataPath:

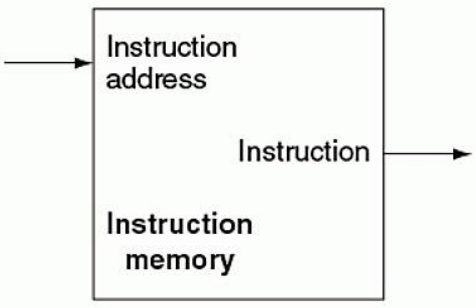
To start, we will look at the datapath elements needed by every instruction.

First, we have *instruction memory*.

Instruction memory is a state element that provides read-access to the instructions of a program and, given an address as input, supplies the corresponding instruction at that address.

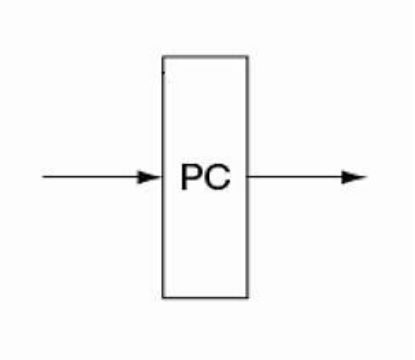
- Code can also be written, e.g., self-modifying code.

Next, we have the *program counter* or *PC*.



The PC is a state element that holds the address of the current instruction. Essentially, it is just a 32-bit register which holds the instruction address and is updated at the end of every clock cycle.

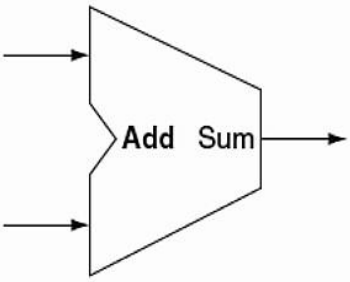
- Normally PC increments sequentially except for branch instructions



The arrows on either side indicate that the PC state element is both readable and writeable.

Lastly, we have the *adder*.

The adder is responsible for incrementing the PC to hold the address of the next instruction.

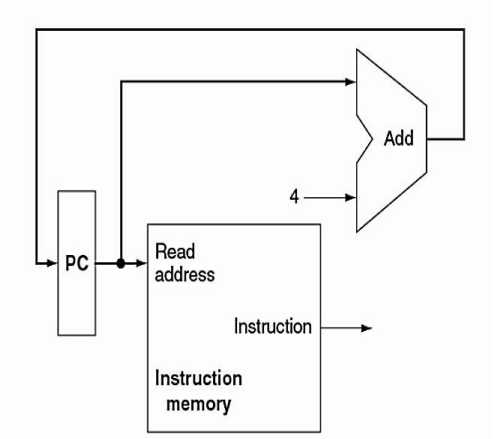


It takes two input values, adds them together and outputs the result.

So now we have instruction memory, PC, and adder datapath elements. Now, we can talk about the general steps taken to execute a program.

- Instruction fetching: use the address in the PC to fetch the current instruction from instruction memory.
- Instruction decoding: determine the fields within the instruction
- Instruction execution: perform the operation indicated by the instruction.
- Update the PC to hold the address of the next instruction.

1. Fetch the instruction at the address in PC.
2. Decode the instruction.
3. Execute the instruction.
4. Update the PC to hold the address of the next instruction.



Note: we perform PC+4 because MIPS instructions are word-aligned.

R-FORMAT INSTRUCTIONS

Now, let's consider R-format instructions. In our limited MIPS instruction set, these are *add*, *sub*, *and*, *or*, and *slt*.

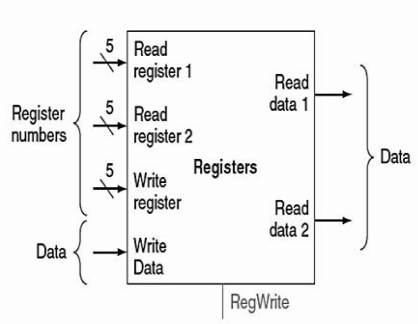
All R-format instructions read two registers, *rs* and *rt*, and write to a register *rd*.

Name	**Fields**					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R format	op	Rs	Rt	Rd	shamt	funct

op – instruction opcode.
rs – first register source operand.
rt – second register source operand.
rd – register destination operand.
shamt – shift amount.
funct – additional opcodes.

To support R-format instructions, we'll need to add a state element called a *register file*. A register file is a collection readable/writeable registers.

- `Read register 1` – first source register. 5 bits wide.
- `Read register 2` – second source register. 5 bits wide.
- `Write register` – destination register. 5 bits wide.
- `Write data` – data to be written to a register. 32 bits wide.



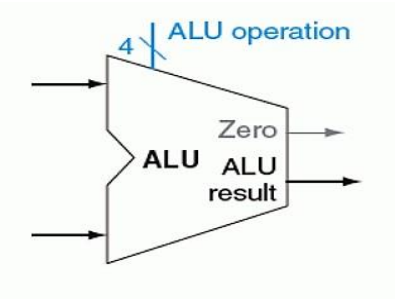
At the bottom, we have the `RegWrite` input. A writing operation only occurs when this bit is set.

The two output ports are:

- `Read data 1` – contents of source register 1.
- `Read data 2` – contents of source register 2.

To actually execute R-format instructions, we need to include the ALU element.

The ALU performs the operation indicated by the instruction. It takes two operands, as well as a 4-bit wide operation selector value. The result of the operation is the output value.



- ALU operation is a part of the control. We discuss datapath first.

We have an additional output specifically for branching – we will cover this in a minute.

Here is our datapath for R- format instructions.

1. Grab instruction address from PC.

2. Fetch instruction from instruction memory.

3. Decode instruction.

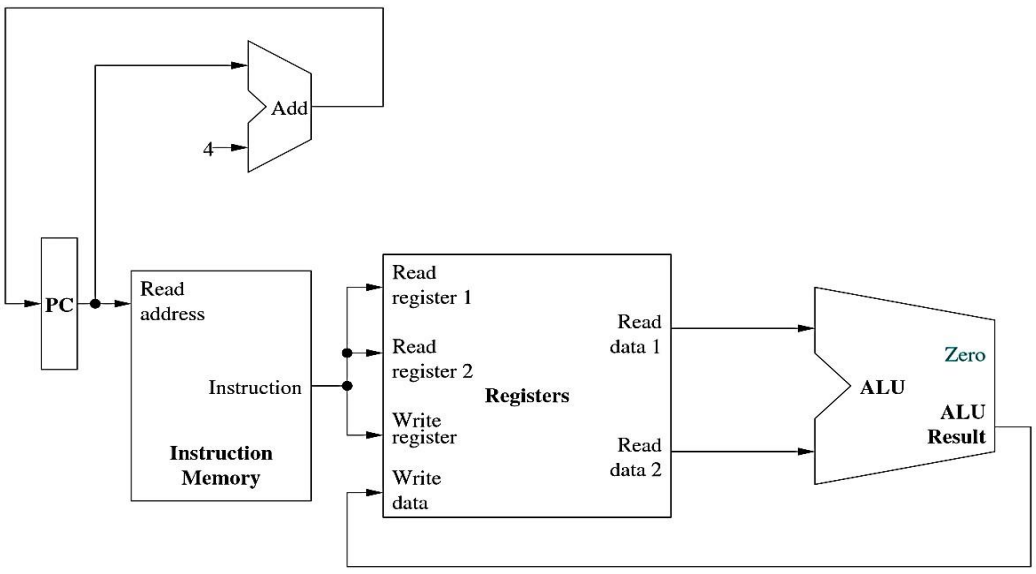
4. Pass rs, rt and rd into read register and write register arguments.

5. Retrieve data from read register 1 and read register 2 (rs and rt).

6. Pass contents of rs and rt into the ALU as operands of the operation to be performed.

7. Retrieve result of operation performed by ALU and pass back as the write data argument of the register file (with the `RegWrite` bit set).

8. Add 4 bytes to the PC value to obtain the word- aligned address of the next instruction.



I-FORMAT INSTRUCTIONS

Now that we have a complete datapath for R-format instructions, let's add in support for I-format instructions. In our limited MIPS instruction set, these are `lw`, `sw`, and `beq`.

- The *op* field is used to identify the type of instruction.
- The *rs* field is the source register.
- The *rt* field is either the source or destination register, depending on the instruction.
- The *immed* field is zero-extended if it is a logical operation. Otherwise, it is signextended.

Name	**Fields**					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
I format	op	rs	Rt	immed		

Data Transfer instructions:

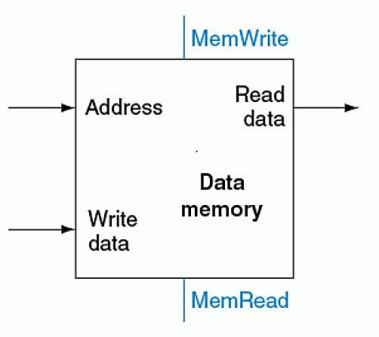
Let's start with accommodating the data transfer. For `lw` and `sw`, we have the following format:

```
lw $rt,immed($rs)
sw $rt, immed($rs)
```

The memory address is computed by *sign-extending* the 16-bit immediate to 32-bits, which is added to the contents of $rs.

In lw, $rt represents the register that will be assigned the memory value. In sw, $rt represents the register whose value will be stored in memory.
Bottom line: we need two more datapath elements to access memory and perform sign-extending.

The *data memory* element implements the functionality for reading and writing data to/from memory.

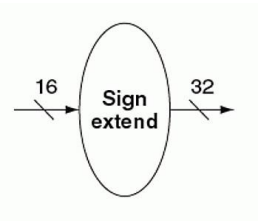


There are two inputs. One for the address of the memory location to access, the other for the data to be written to memory if applicable.

The output is the data read from the memory location accessed, if applicable.

Reads and writes are signaled by MemRead and MemWrite, respectively, which must be asserted for the corresponding action to take place.

To perform sign-extending, we can add a sign extension element.



The sign extension element takes as input a 16-bit wide value to be extended to 32-bits.

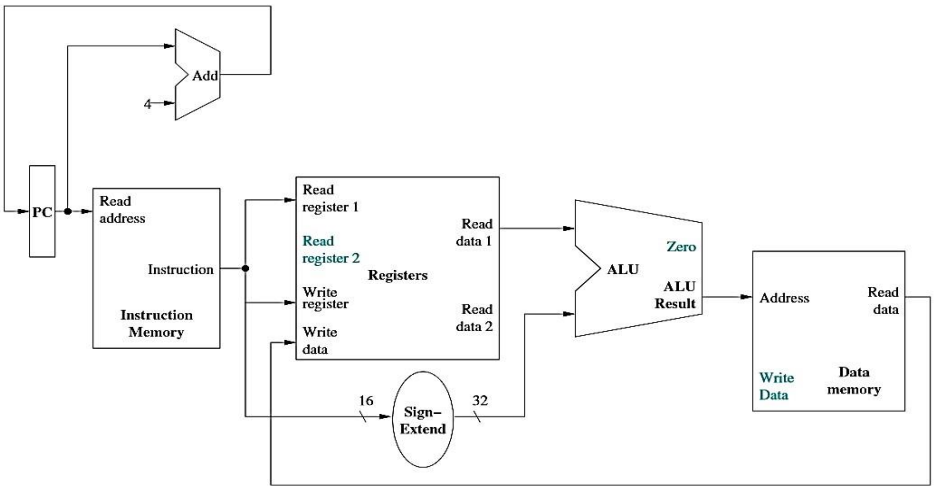
To sign extend, we simply replicate the most-significant bit of the original field until we have reached the desired field width.

For Load instruction

Here, we have modified the datapath to work only for the `lw` instruction.

```
lw $rt, immed($rs)
```

The registers have been added to the datapath for added clarity.

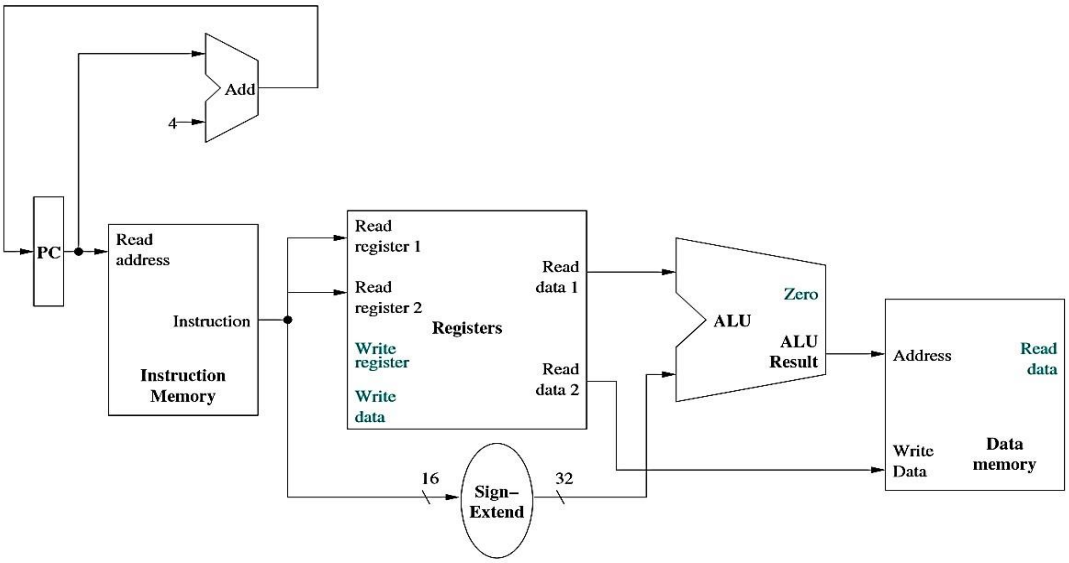


For store instruction:

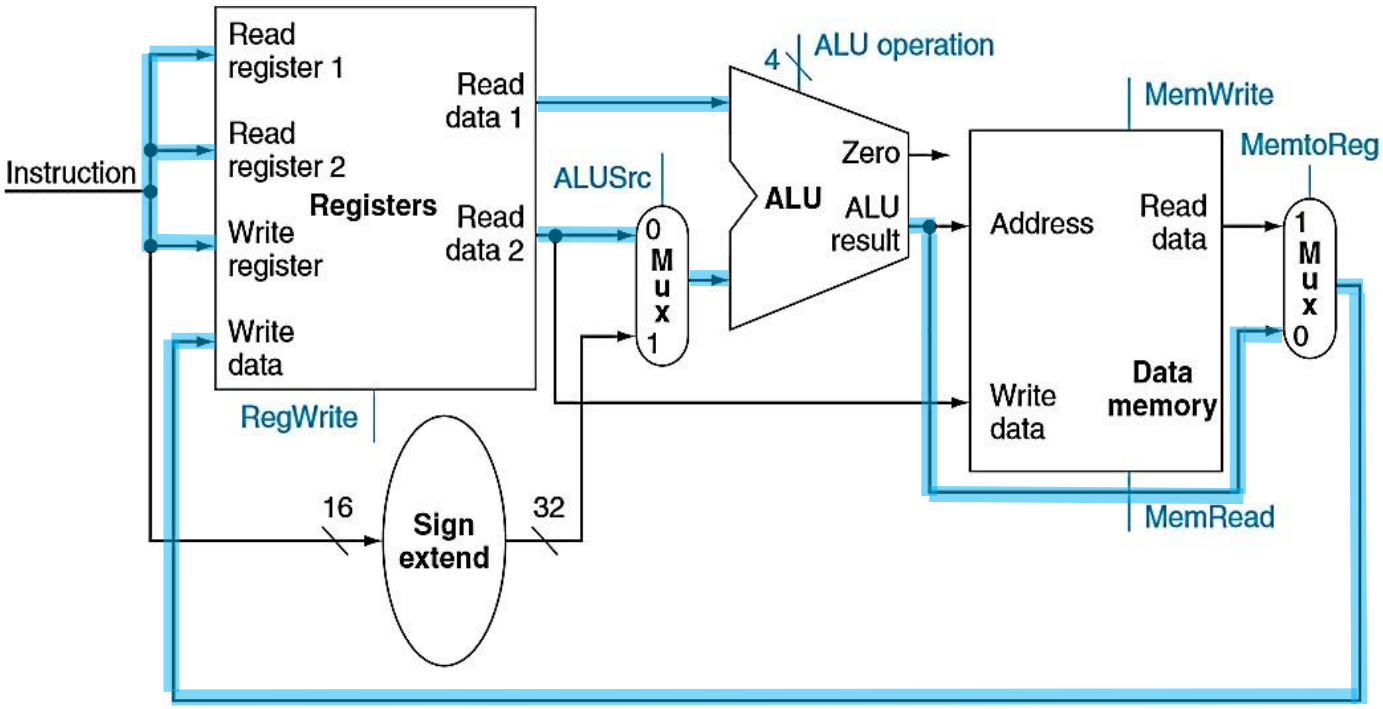
Here, we have modified the datapath to work only for the `sw` instruction.

```
sw $rt, immed($rs)
```

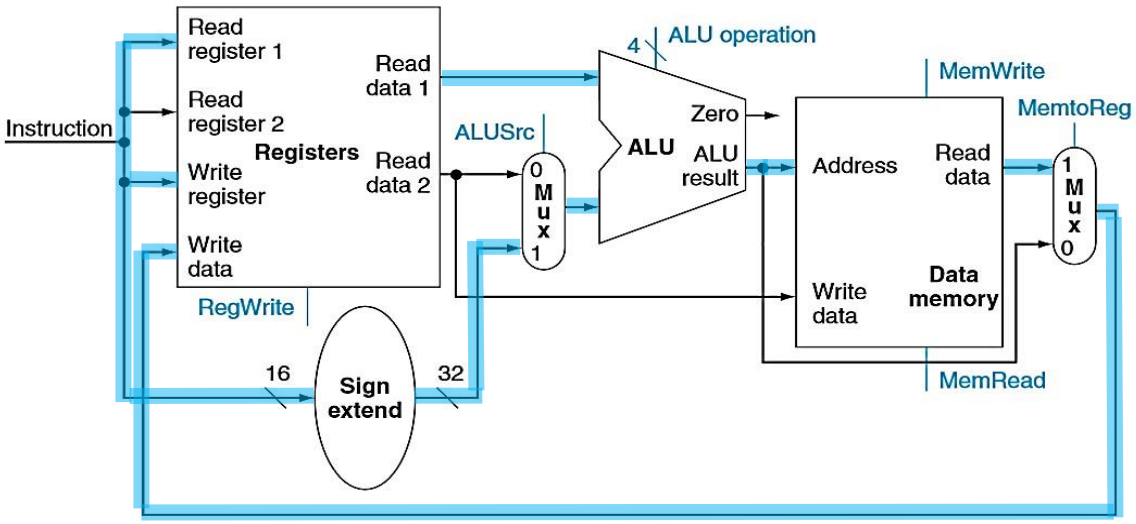
The registers have been added to the datapath for added clarity.



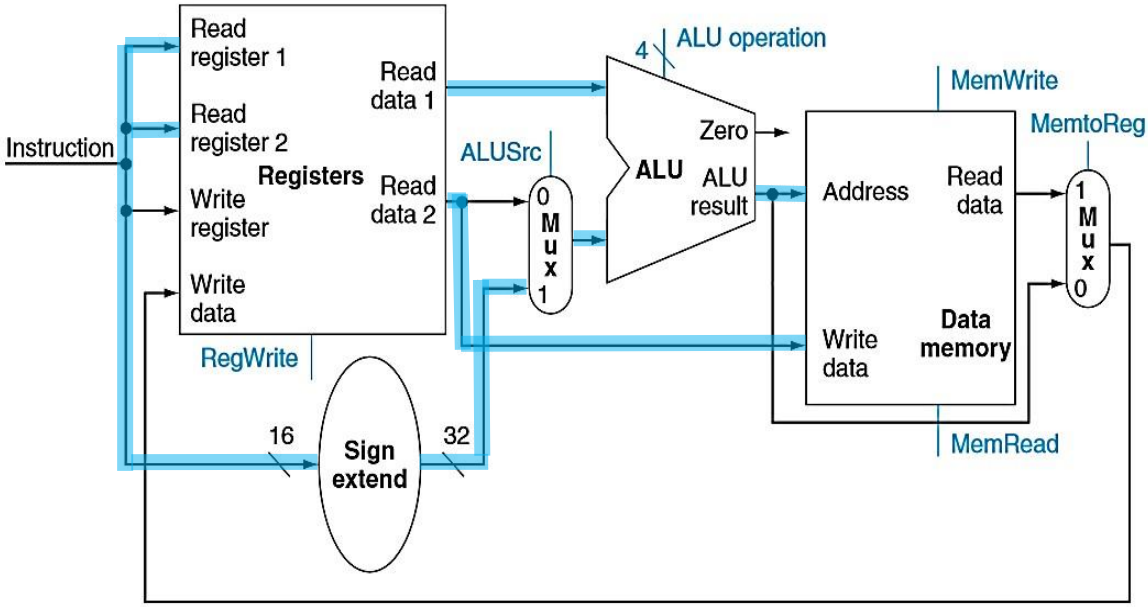
Data Type for Format and Memory Access (add $rd, $rs, $rt)



Data Type for Format and Memory Access (lw $rt, immed ($rs)



Data Type for Format and Memory Access (sw $rt, immed ($rs)



Branch instruction

Now we'll turn out attention to a branching instruction. In our limited MIPS instruction set, we have the beq instruction which has the following form:

```
beq $t1, $t2, target
```

This instruction compares the contents of $t1 and $t2 for equality and uses the 16-bit immediate field to compute the target address of the branch relative to the current address.

Name	Fields					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
I format	op	rs	rt	immed		

Note that our immediate field is only 16-bits so we can't specify a full 32-bit target address. So we have to do a few things before jumping.

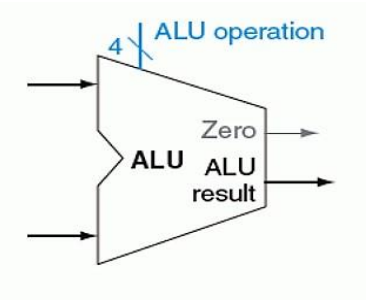
- The immediate field is left-shifted by two because the immediate represents the number of *words* offset from PC+4, not the number of bytes (and we want to get it in number of bytes!).
- We sign-extend the immediate field to 32-bits and add it to PC+4.

```
beq $t1, $t2, target
```

Name	Fields					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
I format	op	rs	rt	immed		

Besides computing the target address, a branching instruction also has to compare the contents of the operands.

As stated before, the ALU has an output line denoted as Zero. This output is specifically hardwired to be set when the result of an operation is zero.



To test whether a and b are equal, we can

- **Set the ALU to perform a subtraction operation.**

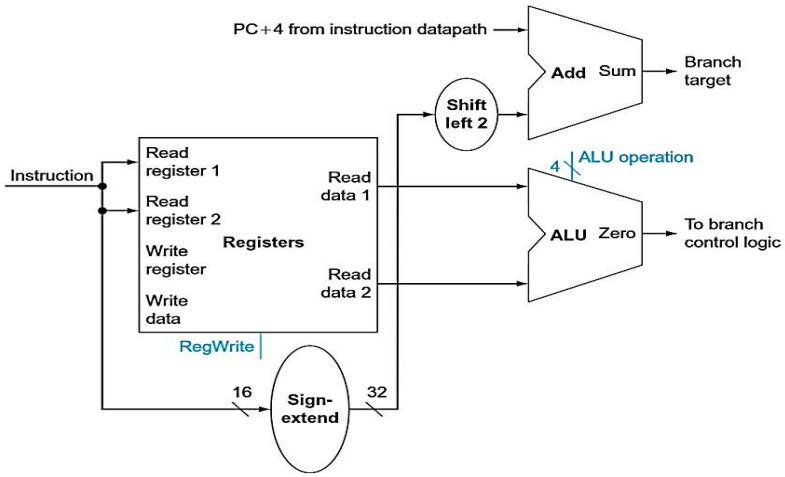
The Zero output line is only set if a – b is 0, indicating a and b are equal.

For branch instruction

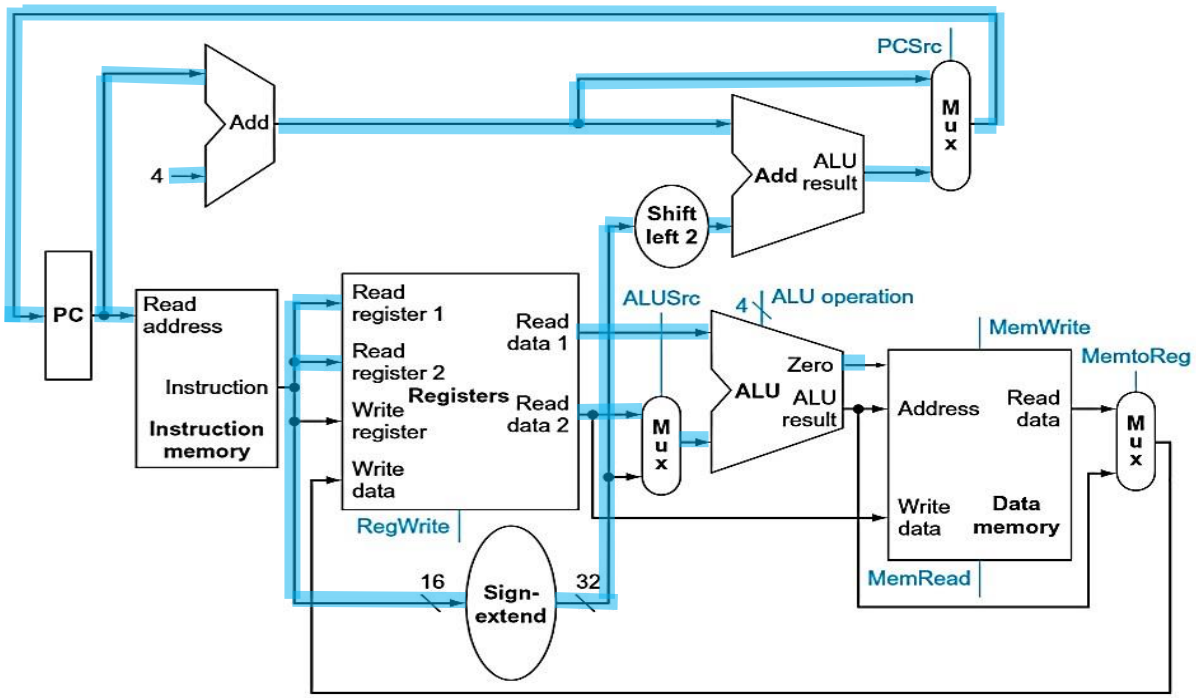
Here, we have modified the datapath to work only for the beq instruction.

```
beq $rs, $rt, immed
```

The registers have been added to the datapath for added clarity.



Now we have a datapath which support all of our R and I format instructions.



J-FORMAT INSTRUCTIONS:

The last instruction we have to implement in our simple MIPS subset is the jump instruction. An example jump instruction is `j L1.` This instruction indicates that the next instruction to be executed is at the address of label L1.

- We have 6 bits for the opcode.
- We have 26 bits for the target address.

Name	Fields					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
J format	Op	target_addr				

Note, we do not have enough space in the instruction to specify a full target address.

- Branching solves this problem by specifying an offset in words.
- Jump instructions solve this problem by specifying *a portion of an absolute address*
- Take the 26-bit target address field of the instruction, left-shift by two (instructions are word-aligned),
- concatenate the result with the upper 4 bits of PC+4.

Here, we have modified the datapath to work only for the `j` instruction.

```
j targaddr
```